EECE5639
COMPUTER VISION

PROJECT 3

STEREO VISION

SHIVA KUMAR DANDE
SAI PRASASTH KOUNDINYA GANDRAKOTA

ABSTRACT

This project creates a stereo correspondence algorithm to generate a dense disparity map from a pair of stereo images. Harris Corner algorithms is used to identify corners and matches are found between the images using NCC and applying a threshold. Then RANSAC is then used to estimate the best fundamental matrix and remove any outliers. The output consists of three images showing the vertical and horizontal disparity components, and a color-coded disparity vector image. The results show that the algorithm accurately estimates the disparity map, providing valuable information for depth estimation and 3D reconstruction.

DESCRIPTION OF ALGORITHMS

HARRIS CORNER DETECTOR

Firstly, Harris Corner Detector algorithm was applied. The input images were converted to greyscale and then both vertical and horizontal Sobel masks were applied to each image. After that, resultant I_x^2, I_y^2 and $I_x * I_y$ values were then convolved with a 3x3 Gaussian Filter. The M matrix was calculated from the final I_x^2, I_y^2 and $I_x * I_y$ values. The Harris R function value was then determined from the M matrix using the formula $R = \det(M) - k*[\text{trace}(M)]^2$ where $\det(M) = \lambda_1 * \lambda_2$, $\text{trace}(M) = \lambda_1 + \lambda_2$ and k value is 0.04. A threshold of 8 was set and the corners were then identified from the R function.

NORMALIZED CROSS CORRELATION

In order to detect the corner matches between the input images the normalized cross correlation matrix was computed between each image pair, using patches of the image with each corner at the center. Pairs of corners with high NCC values were isolated and then applied against a threshold of 0.95 in order to determine the matched corner features between the image pairs.

FUNDAMENTAL MATRIX (RANSAC)

From the corner sets obtained for both images after using NCC and applying a threshold, we then perform the Eight-Point algorithm using a random selection of 8 points from each set. Then we apply RANSAC by calculating the distance between the correspondences and their epipolar lines and determine the number of inliers by applying a max distance of 10, repeating this for around 1000 iterations. The fundamental matrix which produces the best number of inliers is then chosen.

DENSE DISPARITY MAPS

Using the best fundamental matrix obtained, we first convert both images to grayscale and then find the epipolar lines for the left image. We use a search range of 220 and a block size of 9 to compute the image size and block radius and define a search window for every pixel in the left image. Then we define a search window for each pixel in the right image and compute the cost by finding the sum of absolute differences between the two windows. The horizontal and vertical disparity maps are then obtained from disparity of the column with the smallest matching cost. Using these two maps we calculate the hue and saturation for the color disparity map. All disparity values are scaled to the range 0-255.

OUTPUTS

INPUT IMAGES

IMAGE SET 1

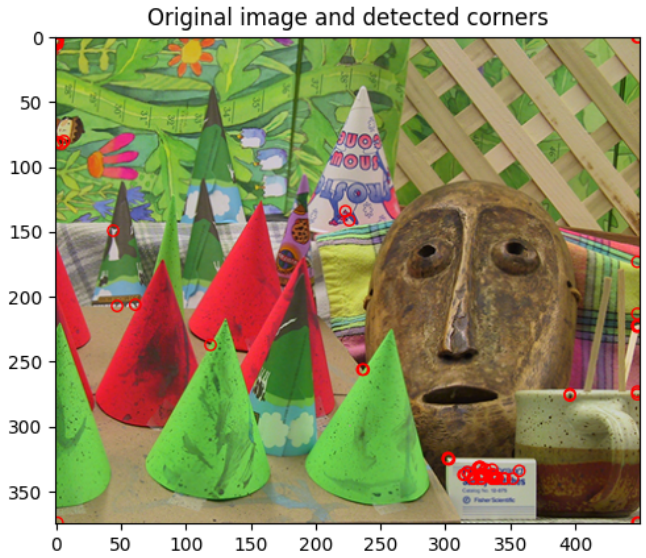
(L) **(R)**

IMAGE SET 2

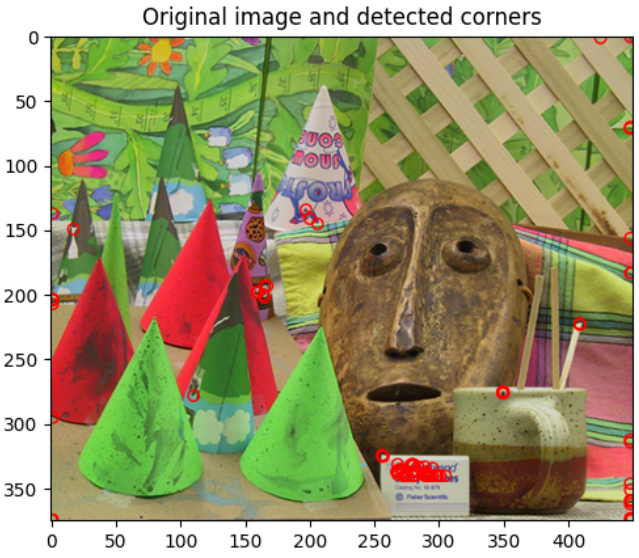
(L) **(R)**

DETECTED CORNERS

IMAGE SET 1

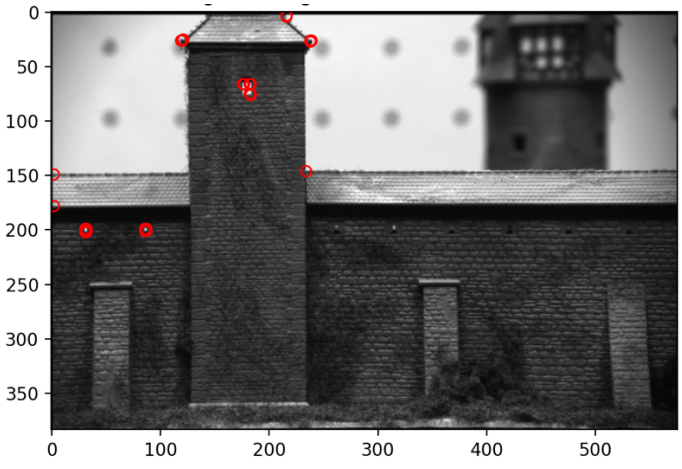


(L)

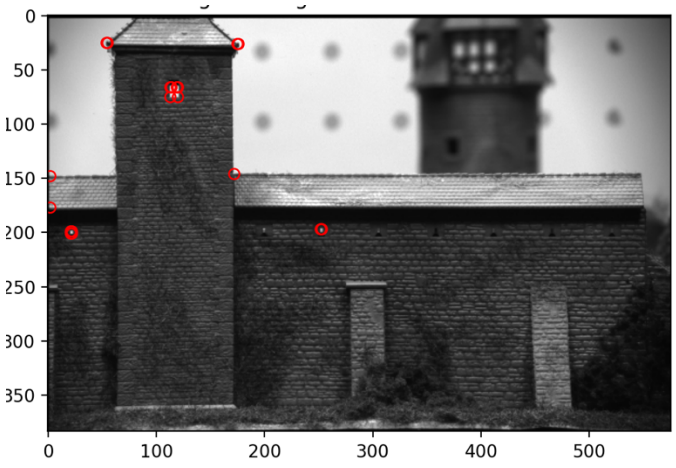


(R)

IMAGE SET 2



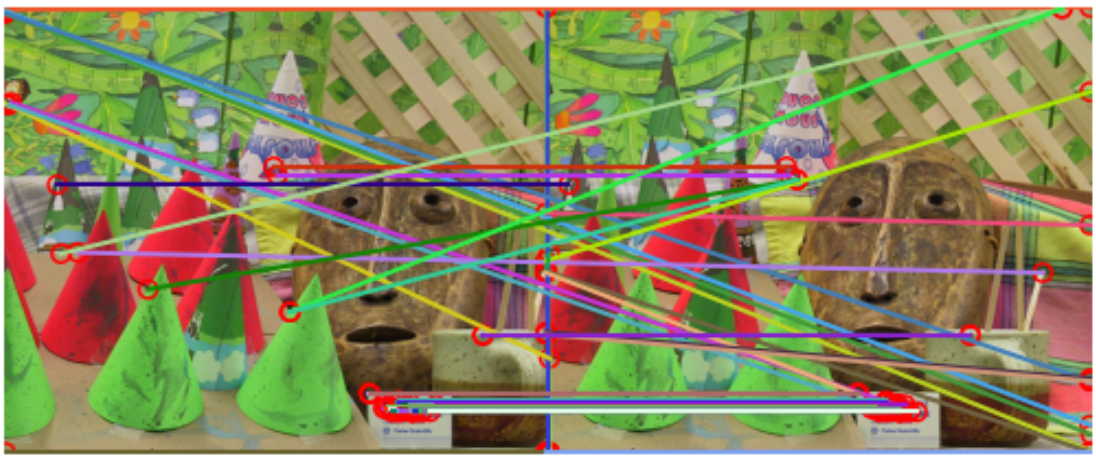
(L)



(R)

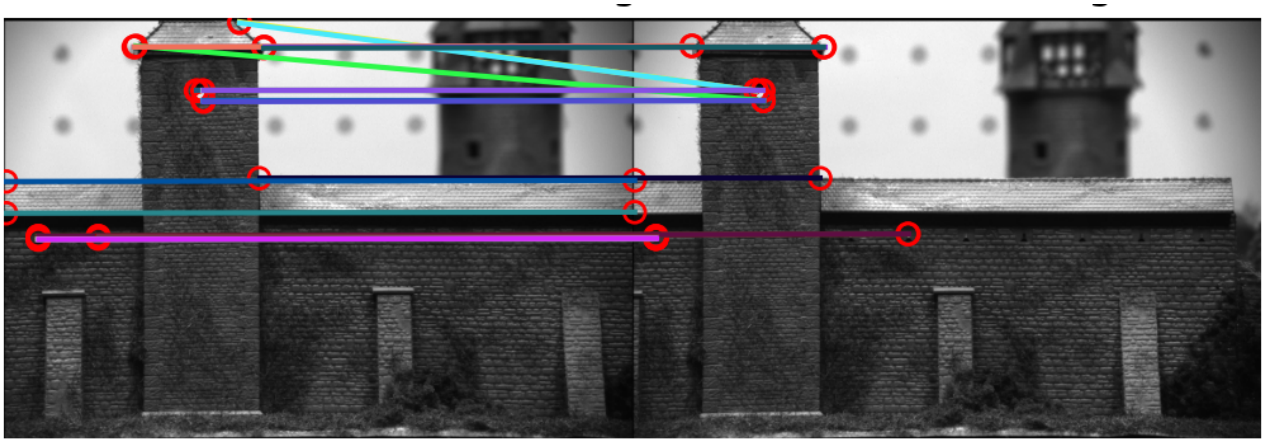
NCC CORNER MATCHES (BEFORE THRESHOLDING)

IMAGE SET 1



(L) **(R)**

IMAGE SET 2



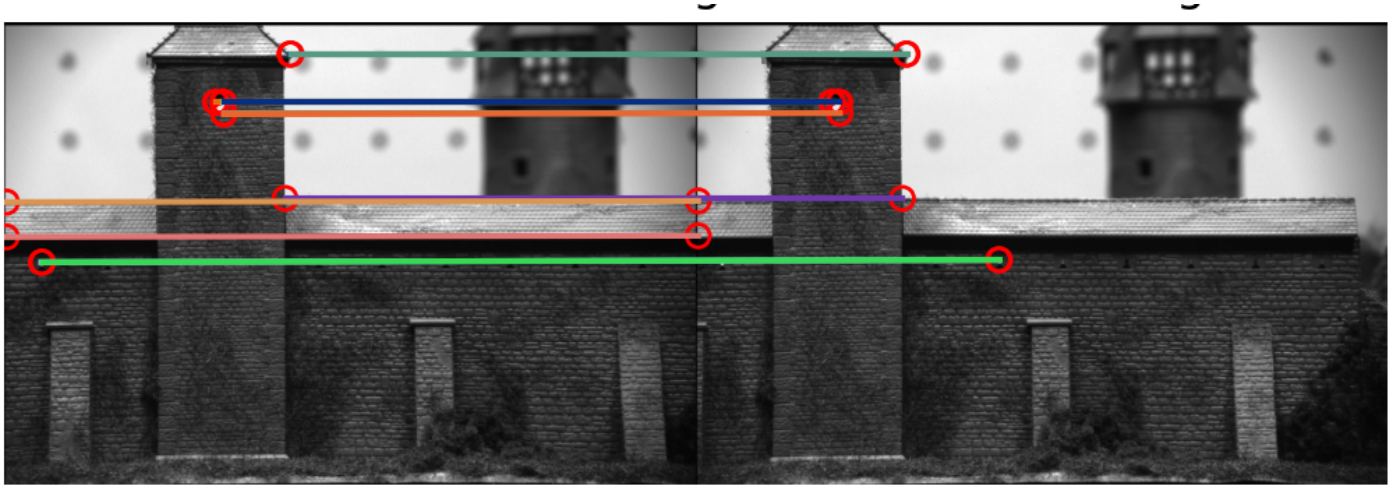
(L) **(R)**

NCC CORNER MATCHES (AFTER THRESHOLDING)

IMAGE SET 1

(L) **(R)**

IMAGE SET 2



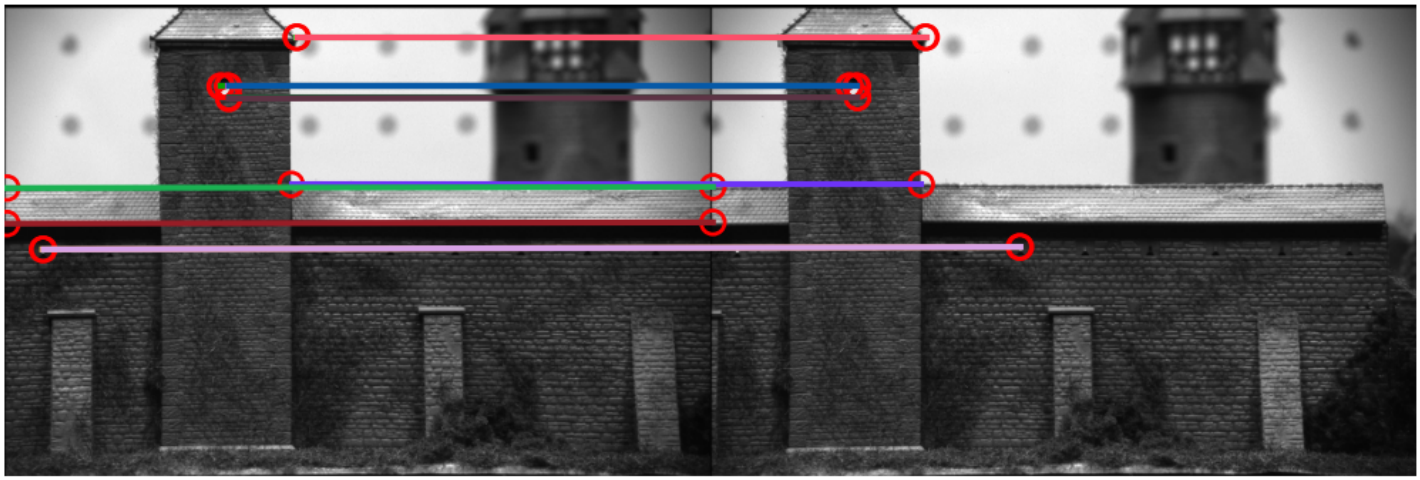
(L) **(R)**

INLIER CORRESPONDENCES (AFTER RANSAC AND FUNDAMENTAL MATRIX ESTIMATION)

IMAGE SET 1

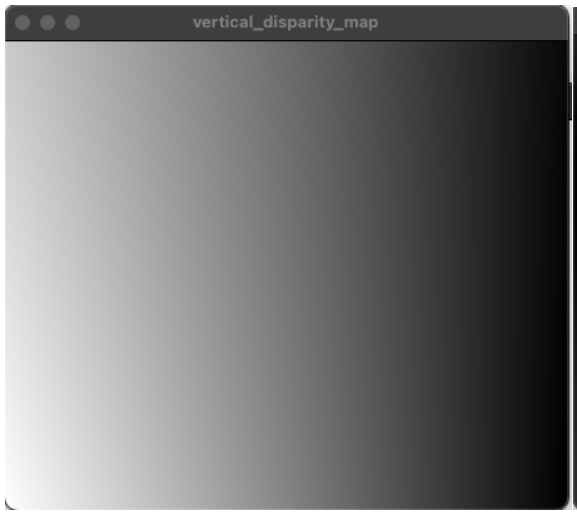
(L) **(R)**

IMAGE SET 2



(L) **(R)**

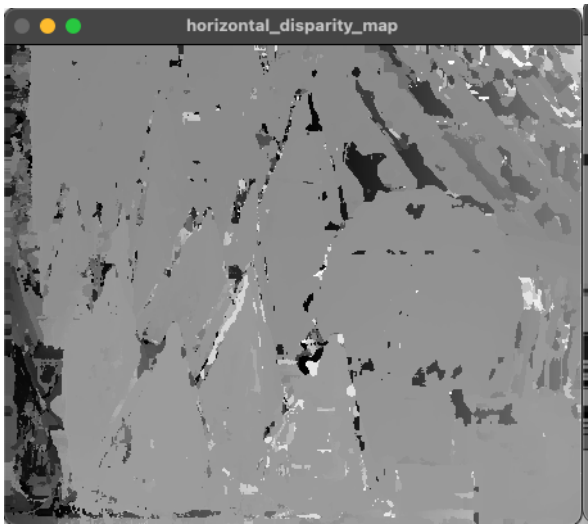
VERTICAL DENSE DISPARITY MAP



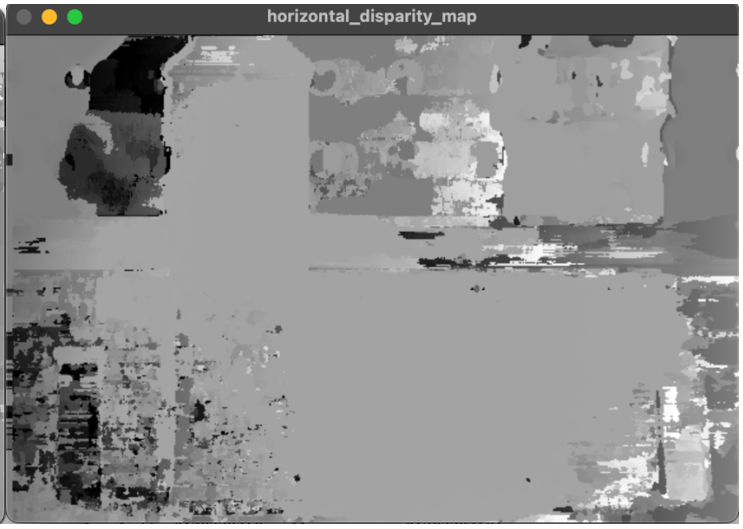
(1)

(2)

HORIZANTAL DENSE DISPARITY MAP

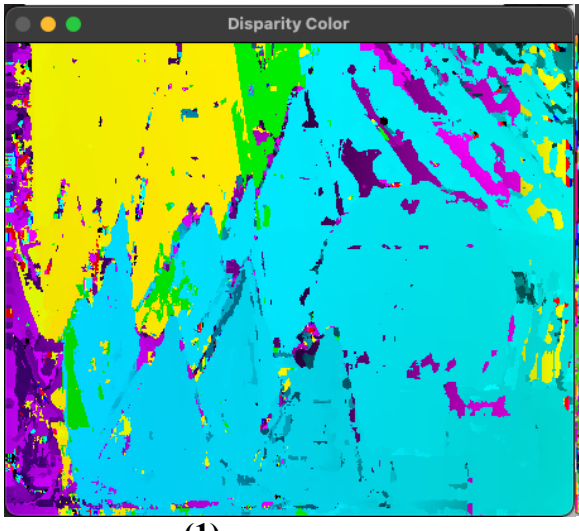


(1)

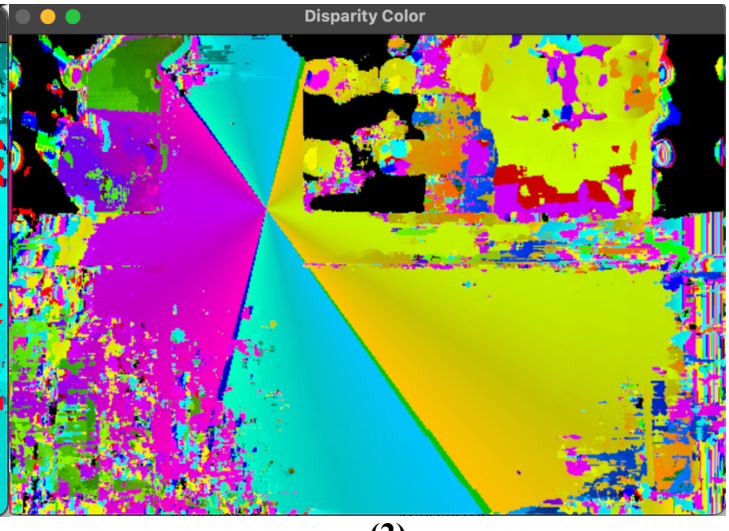


(2)

COLOR DENSE DISPARITY MAP



(1)



(2)

PARAMETERS USED

- In the Harris Corner Detection algorithm, we filter the images with a Sobel Filter to obtain the C matrix and then a Gaussian filter of size 3x3 is applied to the C matrix.
- The threshold for Harris Corner algorithm was 8.
- After finding the corner sets in both images using the computed normalized cross correlation using window patches of size 9x9.
- The number of corner pairs was reduced after the threshold of 0.95 was applied to the NCC values.
- For RANSAC, the maximum number of iterations set was 1000, and after trial and error a max distance of 10 was chosen to classify inliers and outliers.
- For the dense disparity maps, a block size of 9 and a search range of 220 were used to search for points in the right image for each corresponding pixel in the left image.

OBSERVATIONS AND CONCLUSIONS

- In the Harris Corner Detection algorithm, the Gaussian filter applied to the C matrices helps in reducing the noise and improving the robustness of the algorithm.
- The size of the patches being created determines the efficiency and precision of calculating the normalized cross correlation which in turn could lead to unwanted feature matches or instead missed feature matches.
- The threshold for NCC affects the number of potential corner matches in the images. Setting a high value would result in very few matches being detected and risks missing out on features, whereas low threshold values would result in non-features being detected as potential matches as well.
- For RANSAC, as we increase the maximum distance the number of inliers detected increases and ends up giving an unfavorable outcome. However, for low values of the maximum distance we might not be able to determine more than a couple corner points which would result in the process stopping as we need at least 8 correspondences to calculate the Fundamental Matrix.
- By using the Fundamental matrix, you can limit the search space for each pixel's correspondence to the epipolar line in the right image. This significantly reduces the number of potential matches that need to be considered, speeding up the process and improving the accuracy of the disparity map.
- When computing a dense disparity map, the goal is to find the corresponding point for each pixel in the left image within the right image. Without any constraints, you would need to search the entire right image for each pixel in the left image, which is computationally expensive and prone to errors due to ambiguous matches.
- The disparity between the corresponding points in the two images is inversely proportional to the depth of the point in the scene, so by computing a dense disparity map, it's possible to reconstruct the 3D structure of the scene.

APPENDIX

CODE (PYTHON)

```
import numpy as np
from skimage.color import rgb2gray
from skimage.feature import corner_peaks
from skimage.io import imread, imshow
from scipy.signal import convolve2d
from scipy.signal.windows import gaussian
import matplotlib.pyplot as plt
import cv2
from skimage.measure import ransac
from skimage.transform import FundamentalMatrixTransform
import random
# np.random.seed(42)

def compute_rectifying_homography(F):

    # assuming F is your fundamental matrix
    U, S, Vt = np.linalg.svd(F)

    # The left and right singular matrices are the epipolars of image 1 and image 2 respectively
    e1 = U
    e2 = Vt.T


def normalize_points(points):
    """
    Normalize a set of homogeneous points so that the centroid of the points is at the origin,
    and the average distance of the points from the origin is sqrt(2).
    """
    # Compute centroid
    centroid = np.mean(points, axis=0)

    # Shift points so that centroid is at origin
    shifted_points = points - centroid

    # Compute average distance from origin
    mean_distance = np.mean(np.sqrt(shifted_points[:, 0]**2 + shifted_points[:, 1]**2))

    # Scale points so that average distance from origin is sqrt(2)
    scale = np.sqrt(2) / mean_distance
    T = np.array([[scale, 0, -scale*centroid[0]],
                  [0, scale, -scale*centroid[1]],
                  [0, 0, 1]])
    normalized_points = np.dot(T, points.T).T

    return normalized_points, T
```

```
def compute_fundamental_matrix(corners_image1, corners_image2):
    """
    Compute the fundamental matrix using the 8-point algorithm.
    """
    # Normalize the correspondences
    x1, T1 = normalize_points(corners_image1)
    x2, T2 = normalize_points(corners_image2)

    # Construct the A matrix
    A = np.zeros((corners_image1.shape[0], 9))
    for i in range(corners_image1.shape[0]):
        A[i] = [x1[i, 0]*x2[i, 0], x1[i, 0]*x2[i, 1], x1[i, 0],
                x1[i, 1]*x2[i, 0], x1[i, 1]*x2[i, 1], x1[i, 1],
                x2[i, 0], x2[i, 1], 1]

    # Compute the singular value decomposition of A
    U, S, Vt = np.linalg.svd(A)

    # The fundamental matrix is the column of V corresponding to the smallest singular value
    F = np.reshape(Vt[-1], (3, 3))

    # Enforce rank 2 constraint on F
    U, S, Vt = np.linalg.svd(F)
    S[2] = 0
    F = np.dot(U, np.dot(np.diag(S), Vt))

    # Unnormalize the fundamental matrix
    F = np.dot(T2.T, np.dot(F, T1))
    # x1 = corners_image1
    # x2 = corners_image2
    # # Construct the A matrix
    # A1 = np.zeros((corners_image1.shape[0], 9))
    # for i in range(corners_image1.shape[0]):
    #     A1[i] = [x1[i, 0]*x2[i, 0], x1[i, 0]*x2[i, 1], x1[i, 0],
    #              x1[i, 1]*x2[i, 0], x1[i, 1]*x2[i, 1], x1[i, 1],
    #              x2[i, 0], x2[i, 1], 1]

    # # Compute the singular value decomposition of A
    # U1, S1, Vt1 = np.linalg.svd(A1)

    # # The fundamental matrix is the column of V corresponding to the smallest singular value
    # F1 = np.reshape(Vt1[-1], (3, 3))

    # # Enforce rank 2 constraint on F
    # U1, S1, Vt1 = np.linalg.svd(F1)
    # S1[2] = 0
    # F1 = np.dot(U1, np.dot(np.diag(S1), Vt1))
    # print(F1)
```

```
    return F

def ransac_fundamental_matrix(corners_image1, corners_image2, n_iters, thresh):
    """
    Find the best Fundamental matrix using RANSAC.
    """
    max_inliers = []
    best_F = None

    ones_array = np.ones(corners_image1.shape[0])
    corners_image1 = np.column_stack((corners_image1, ones_array)).astype(np.int64)
    corners_image2 = np.column_stack((corners_image2, ones_array)).astype(np.int64)

    for i in range(n_iters):
        # Randomly select 8 correspondences
        inds = random.sample(range(corners_image1.shape[0]), 8)
        corners_image1_8 = corners_image1[inds, :]
        corners_image2_8 = corners_image2[inds, :]
        correspondences = np.column_stack((corners_image1, corners_image2)).astype(np.int64)

        # Compute the Fundamental matrix
        F = compute_fundamental_matrix(corners_image1_8, corners_image2_8)

        # Compute the distances between the correspondences and the epipolar lines
        x1 = corners_image1
        x2 = corners_image2
        l2 = np.dot(F, x1.T)
        l1 = np.dot(F.T, x2.T)
        d2 = np.abs(np.sum(x2*l2.T, axis=1)) / np.sqrt(l2[0]**2 + l2[1]**2)
        d1 = np.abs(np.sum(x1*l1.T, axis=1)) / np.sqrt(l1[0]**2 + l1[1]**2)
        distances = d1 + d2
        # Count the number of inliers
        inliers = correspondences[distances < thresh]

        # Keep track of the largest set of inliers
        if len(inliers) > len(max_inliers):
            max_inliers = inliers
            best_F = compute_fundamental_matrix(inliers[:, :3], inliers[:, 3:])

    return best_F, max_inliers

def threshold_ncc(max_ncc_values, matched_corner_sets, threshold):
    ncc_th = []
    matched_corner_sets_th = []

    for i in range(len(max_ncc_values)):
        if max_ncc_values[i] >= threshold:
            ncc_th.append(max_ncc_values[i])
            matched_corner_sets_th.append(matched_corner_sets[i, :])
```

```
    return np.array(ncc_th), np.array(matched_corner_sets_th)

def create_image_patch(row_center, col_center, image, w_size):
    img_size = image.shape
    start_i = max(row_center - (w_size - 1) // 2, 0)
    start_j = max(col_center - (w_size - 1) // 2, 0)
    end_i = min(row_center + (w_size - 1) // 2 + 1, img_size[0])
    end_j = min(col_center + (w_size - 1) // 2 + 1, img_size[1])

    if len(image.shape) == 3:
        gray_image = rgb2gray(image)
    else:
        gray_image = image

    image_patch = gray_image[start_i:end_i, start_j:end_j]
    image_patch = np.pad(image_patch, ((w_size - image_patch.shape[0], 0), (w_size - image_patch.shape[1],
0)), mode='constant', constant_values=0)
    return image_patch.astype(np.float64)

def norm_cross_corr(row_col1, image1, row_col2, image2):
    w_size = 9
    max_ncc_values = []
    matched_corner_sets = []

    for row_center_img1, col_center_img1 in row_col1:
        image1_patch = create_image_patch(row_center_img1, col_center_img1, image1, w_size)
        mean_img1_patch = np.mean(image1_patch)

        max_ncc = 0
        img2_corner = np.zeros(2)

        for row_center_img2, col_center_img2 in row_col2:
            image2_patch = create_image_patch(row_center_img2, col_center_img2, image2, w_size)
            mean_img2_patch = np.mean(image2_patch)

            product = (image1_patch - mean_img1_patch) * (image2_patch - mean_img2_patch)
            numerator_term = np.sum(product)

            s1 = np.sum((image1_patch - mean_img1_patch) ** 2)
            s2 = np.sum((image2_patch - mean_img2_patch) ** 2)
            denominator_term = np.sqrt(s1 * s2)

            ncc = numerator_term / denominator_term

            if max_ncc < ncc:
                img1_corner = np.array([row_center_img1, col_center_img1])
                img2_corner = np.array([row_center_img2, col_center_img2])
                max_ncc = ncc
```

```
            max_ncc_values.append(max_ncc)
            matched_corner_sets.append(np.hstack((img1_corner, img2_corner)))

    return np.array(max_ncc_values), np.array(matched_corner_sets)

def harris_corner_alg(image):
    if len(image.shape) == 3:
        gray_image = rgb2gray(image)
    else:
        gray_image = image

    sobel_X = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
    sobel_Y = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])

    Ix = convolve2d(gray_image, sobel_X, mode='same')
    Iy = convolve2d(gray_image, sobel_Y, mode='same')

    size_window = 3
    gaussian_filter = gaussian(size_window**2, 1).reshape(size_window, size_window)
    Ix2 = convolve2d(Ix**2, gaussian_filter, mode='same')
    Iy2 = convolve2d(Iy**2, gaussian_filter, mode='same')
    Ixy = convolve2d(Ix * Iy, gaussian_filter, mode='same')

    det = Ix2 * Iy2 - Ixy * Ixy
    trace = Ix2 + Iy2
    k = 0.04
    R = det - k * (trace**2)

    size_window = 11
    threshold = 8
    median_filtered = np.maximum(R, np.zeros_like(R))

    harris = (R == median_filtered) & (R > threshold)
    r, c = np.where(harris >= 1)

    plt.figure()
    plt.imshow(image, cmap='gray')
    plt.scatter(c, r, marker='o', edgecolors='r', facecolors='none')
    plt.title('Original image and detected corners')
    plt.show()
    return r, c

# Usage example:
path = "/Users/prasasth/Downloads/cvp3/cast-left-1.jpeg"
image1 = imread(path)
path = "/Users/prasasth/Downloads/cvp3/cast-right-1.jpeg"
image2 = imread(path)
```

```
r, c = harris_corner_alg(image1)
row_col1 = np.column_stack((r, c))
r, c = harris_corner_alg(image2)
row_col2 = np.column_stack((r, c))

max_ncc, matched_corner_sets = norm_cross_corr(row_col1, image1, row_col2, image2)
corners_image1 = matched_corner_sets[:, 0:2]
corners_image2 = matched_corner_sets[:, 2:4]

n = matched_corner_sets.shape[0]  # number of matches
matches = np.column_stack((np.arange(n), np.arange(n)))

from skimage.feature import plot_matches
fig, ax = plt.subplots(1, 1)
plot_matches(ax, image1, image2, corners_image1, corners_image2, matches=matches, keypoints_color='r',
matches_color=None, only_matches=False)
ax.axis('off')
ax.set_title('Corner matches between the two images before thresholding the NCC Values')
plt.show()

ncc_threshold = 0.95
thresholded_Ncc, thresholded_Matches = threshold_ncc(max_ncc, matched_corner_sets, ncc_threshold)

corners_image1 = thresholded_Matches[:, 0:2]
corners_image2 = thresholded_Matches[:, 2:4]

n = thresholded_Matches.shape[0]  # number of matches
matches = np.column_stack((np.arange(n), np.arange(n)))

from skimage.feature import plot_matches
fig, ax = plt.subplots(1, 1)
plot_matches(ax, image1, image2, corners_image1, corners_image2, matches=matches, keypoints_color='r',
matches_color=None, only_matches=False)
ax.axis('off')
ax.set_title('Corner matches between the two images after thresholding the NCC Values')
plt.show()
# hiiiii

best_Fundamental_matrix, inliers = ransac_fundamental_matrix(corners_image1, corners_image2, 1000, 10)
print("Fundamental Matrix:\n", best_Fundamental_matrix)
print("Inliers : \n", inliers)

# Visualize the inlier correspondences
n = inliers.shape[0]
matches = np.column_stack((np.arange(n), np.arange(n)))

fig, ax = plt.subplots(1, 1)
plot_matches(ax, image1, image2, inliers[:, 0:2], inliers[:, 3:5], matches=matches, keypoints_color='r',
matches_color=None, only_matches=False)
```

```
ax.axis('off')
ax.set_title('Inlier correspondences after RANSAC and Fundamental Matrix estimation')
plt.show()


import numpy as np
import cv2
from skimage.color import rgb2gray, gray2rgb, hsv2rgb
from skimage.transform import ProjectiveTransform

def compute_dense_disparity_map(image_left, image_right, fundamental_matrix, block_size=9,
search_range=220):
    # Convert images to grayscale
    gray_left = rgb2gray(image_left)
    gray_right = rgb2gray(image_right)

    # Compute image size and block radius
    h, w = gray_left.shape
    r = block_size // 2

    # Compute the epipolar lines for each pixel in the left image
    pts_left = np.array([(x, y) for y in range(h) for x in range(w)])
    print("pts_left : \n", pts_left)
    print("pts_left.reshape(-1, 1, 2) : ", pts_left.reshape(-1, 1, 2))
    epipolar_lines = cv2.computeCorrespondEpilines(pts_left.reshape(-1, 1, 2), 1, fundamental_matrix)
    print(epipolar_lines)
    # Compute the disparity maps
    horizontal_disparity_map = np.zeros((h, w))
    vertical_disparity_map = np.zeros((h, w))
    color_disparity_map = np.zeros((h, w, 3))

    for i, pt_left in enumerate(pts_left):
        # Define the search window for the pixel in the left image
        x_left, y_left = pt_left
        window_left = gray_left[max(0, y_left-r):min(h, y_left+r+1), max(0, x_left-r):min(w, x_left+r+1)]

        # Compute the range of columns in the right image to search
        search_min = max(0, x_left - search_range)
        search_max = min(w, x_left + search_range)

        # Compute the matching cost for each column in the search range
        costs = []
        for x_right in range(search_min, search_max):
            # Define the search window for the pixel in the right image
            window_right = gray_right[max(0, y_left-r):min(h, y_left+r+1), max(0, x_right-r):min(w, x_right+r+1)]

            # Adjust window size to match smaller window
            window_size = min(window_left.shape[0], window_right.shape[0]), min(window_left.shape[1],
window_right.shape[1])
```

```
            window_left = window_left[:window_size[0], :window_size[1]]
            window_right = window_right[:window_size[0], :window_size[1]]

            # Compute the sum of absolute differences (SAD) between the two windows
            cost = np.sum(np.abs(window_left - window_right))

            # Add the cost to the list
            costs.append(cost)

        # Find the column in the search range with the smallest matching cost
        best_match = np.argmin(costs)
        disparity = x_left - (search_min + best_match)

        # Set the disparity values in the disparity maps
        horizontal_disparity_map[y_left, x_left] = disparity
        vertical_disparity_map[y_left, x_left] = epipolar_lines[i][0][1]

        # Set the disparity values in the color disparity map
        hue = 180 * (np.arctan2(epipolar_lines[i][0][1], disparity) + np.pi) / (2 * np.pi)
        saturation = np.sqrt(disparity**2 + epipolar_lines[i][0][1]**2)
        color_disparity_map[y_left, x_left] = [hue, saturation, 1]

    # Convert the color disparity map from HSV to RGB
    color_disparity_map = hsv2rgb(color_disparity_map)

    # Scale the disparity maps so the lowest disparity is 0 and the highest disparity is 255
    horizontal_disparity_map = 255 * (horizontal_disparity_map - np.min(horizontal_disparity_map)) /
(np.max(horizontal_disparity_map) - np.min(horizontal_disparity_map))
    vertical_disparity_map = 255 * (vertical_disparity_map - np.min(vertical_disparity_map)) /
(np.max(vertical_disparity_map) - np.min(vertical_disparity_map))

    # Return the disparity maps
    return horizontal_disparity_map, vertical_disparity_map, color_disparity_map

horizontal_disparity_map, vertical_disparity_map, color_disparity_map =
compute_dense_disparity_map(image1, image2, best_Fundamental_matrix)
# cv2.imshow('Left Image', image_left_rect)
# cv2.imshow('Right Image', image_right_rect)
print(horizontal_disparity_map)
print(vertical_disparity_map)
print(color_disparity_map)

horizontal_disparity_map_8bit = np.uint8(horizontal_disparity_map)
vertical_disparity_map_8bit = np.uint8(vertical_disparity_map)
color_disparity_map_8bit = np.uint8(color_disparity_map)
cv2.imshow('horizontal_disparity_map', horizontal_disparity_map_8bit)
cv2.waitKey(0)
cv2.destroyAllWindows()
cv2.imshow('vertical_disparity_map', vertical_disparity_map_8bit)
```

```
cv2.waitKey(0)
cv2.destroyAllWindows()
cv2.imshow('Disparity Color', color_disparity_map_8bit)
cv2.waitKey(0)
cv2.destroyAllWindows()
```